Docket No. MS1-1800US / 301447.2

SYSTEM, METHOD, AND API FOR PROGRESSIVELY INSTALLING A SOFTWARE APPLICATION

EV316935295

Related Applications

This application is a Continuation-in-Part of pending U.S. Patent Application No. 10/444699, filed on 5/22/03, entitled SYSTEM AND METHOD FOR PROGRESSIVELY INSTALLING A SOFTWARE APPLICATION in the names of Mark Alcazar, Michael Dunn, Adriaan Carter, and Prasad Tammana, incorporated by reference herein.

Background of the Invention

There are two main types of applications available today. A first type of application is a client-side application. The client-side application resides on a client computer and is available for use whenever the client computer is operational. This client-side application undergoes a distinct installation state before it is available for use. Typically, the installation state displays some form of a progress user-interface, such as a thermometer, during installation. During the installation state, the client-side application is not available for use. The client-side application must be fully installed before a user can use the application.

The other type of application is commonly referred to as a Web application or Web app. The Web app is stored on a Web server. The Web app is commonly deployed as multiple Web pages accessible over the Internet. A conventional Web app includes multiple Web pages representing markup-based documents. The Web app may also include scripts or other resources that are accessed through the Web pages. For most Web apps, the multiple Web pages and resources are hyperlinked together in such a way that the "business logic" of the Web app is distributed over the multiple resources. Each page is responsible for a portion of the overall business logic, and, by navigating from page to page, the user can experience the entire Web app. For the purpose of this document, the term "navigating" refers to causing a hosting environment to retrieve a resource associated with the Web app, such

as by activating a hyperlink. Navigating to a resource typically involves navigating away from another resource where the navigated-to resource is the one being retrieved by the hosting environment. Web apps do not require an installation phase and are not available once the client computer is disconnected from the Web server.

Both of these methods for interacting with a software application have advantages and disadvantages, neither one is ideal.

Summary of the Invention

The present invention provides a system and method for progressively installing a software application so that a user may begin interacting with the application immediately. Then, while interacting with the application, the application may be progressively installed on the user's computer, and if desired, become available offline at a later time. The progressive installation includes three states: a startup state, a demand state, and a final state. None of the states require a dedicated installation phase in which the application is unavailable for use. Instead, the progressive installation of the present invention blends the two forms of application installation in a manner such that the Web app may be interacted with as a conventional Web app, and then smoothly transitioned to a client-side application without impacting the user's interaction with the application.

The invention provides a mechanism for progressively installing an application. The progressive installation transitions through three states: a start-up state, a demand state, and an install state. During the start-up state, a subset of components associated with the application is downloaded and stored in a local data store. The subset is sufficient to allow execution of the application in a manner similar to a web application. During the demand state, additional resource associated with the application are downloaded upon activation of a hyperlink on a Web page associated with the application. The additional resources that are on demand resources are stored in the local data store. The additional resources that are online resources are stored in a transient cache. During the installed state, the application executes in a manner similar to a client-side application. Transitioning from the demand state to the installed state

occurs without impacting a user's interaction with the application. The transition may occur autonomously based on the number of additional resources stored in the local data store or upon an external trigger. During the transition, additional resources that have not been previously downloaded are downloaded to the local data store. In addition, state derived during the demand state is saved with the application, which allows the application to resume from the same state when executed offline.

Brief Description of the Drawings

FIGURE 1 illustrates an exemplary computing device that may be used in one exemplary embodiment of the present invention.

FIGURE 2 is a functional block diagram overview of a distributed networking environment in which implementations of the invention may be embodied.

FIGURE 3 is an illustrative screen display that may be presented by Web browsing software for enabling the progressive download of an application, in accordance with one implementation of the invention.

FIGURE 4 is a state diagram illustrating various states of the progressive installation of an application, in accordance with one implementation of the invention.

FIGURE 5 is a logical flow diagram generally illustrating a process during a start-up state of the progressive installation.

FIGURE 6 is a logical flow diagram generally illustrating a process during a demand state of the progressive installation.

FIGURE 7 is a logical flow diagram generally illustrating a process for transitioning between the demand state and an install state of the progressive installation.

FIGURES 8-10 are a series of block diagrams graphically illustrating files that are loaded during the progressive installation, in accordance with one implementation of the invention.

Detailed Description of the Preferred Embodiment

Briefly, the present invention provides a system and method for progressively installing a software application so that a user may begin interacting with the application immediately. Then, while interacting with the application, the application may be progressively installed on the user's computer, and, if desired, installed in a manner such that the application is available offline at a later time. The progressive installation includes three states: a startup state, a demand state, and a final state. None of the states require a dedicated installation phase in which the application is unavailable for use. Instead, the progressive installation of the present invention blends the two forms of application installation in a manner such that the Web app may be interacted with as a conventional Web app, and then provides a mechanism for smoothly transitioning the application to an offline application without impacting the user's interaction with the application.

Exemplary Operating Environment

FIGURE 1 illustrates an exemplary computing device that may be used in one exemplary embodiment of the present invention. FIGURE 1 illustrates an exemplary computing device that may be used in one exemplary embodiment of the present invention. In a very basic configuration, computing device **100** typically includes at least one processing unit **102** and system memory **104**. Depending on the exact configuration and type of computing device, system memory **104** may be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. System memory **104** typically includes an operating system **105**, one or more program modules **106**, and may include program data **107**. This basic configuration is illustrated in FIGURE 1 by those components within dashed line **108**.

Computing device **100** may have additional features or functionality. For example, computing device **100** may also include additional data storage devices (removable and/or non-removable) such as, for example, magnetic disks, optical disks, or tape. Such additional storage is illustrated in FIGURE 1 by removable storage **109** and non-removable storage **110**. Computer storage media may include volatile and nonvolatile, removable and non-removable media implemented in any method or

technology for storage of information, such as computer readable instructions, data structures, program modules, or other data. System memory **104**, removable storage **109** and non-removable storage **110** are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by computing device **100**. Any such computer storage media may be part of device **100**. Computing device **100** may also have input device(s) **112** such as keyboard, mouse, pen, voice input device, touch input device, etc. Output device(s) **114** such as a display, speakers, printer, etc. may also be included. These devices are well know in the art and need not be discussed at length here.

Computing device **100** may also contain communication connections **116** that allow the device to communicate with other computing devices **118**, such as over a network. Communication connections **116** is one example of communication media. Communication media may typically be embodied by computer readable instructions, data structures, program modules, or other data in a modulated data signal, such as a carrier wave or other transport mechanism, and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. The term computer readable media as used herein includes both storage media and communication media.

Exemplary Networked Environment

FIGURE 2 is a functional block diagram overview of a distributed networking environment in which implementations of the invention may be embodied. As illustrated in FIGURE 2, two or more computers, such as a server **202** and a client computer **220**, are connected over a network **205**. Server **202** and client computer **220** are computing devices such as the one described above in conjunction with FIGURE 1.

The computers may be connected in a corporate environment, where the network **205** may be a local area network or a wide area network. Similarly, the computers may be arbitrarily connected over a wide area network, such as the Internet.

The server **202** is a computing device that is configured to make resources available to other computing devices connected to the network **205**. The server **202** may include Web serving software to serve Internet related resources, such as HyperText Markup Language (HTML) documents and the like. The server **202** includes local storage in the form of a server data store **210**. On the server data store **210** are at least some of the resources made available by the server **202** over the network **205**. In particular, a deployment manifest **212** is stored on the server data store **210**, as well as an application package **214** and additional application resources **216**, which are described in detail later in conjunction with FIGURES 8-10. The server **202** also includes other applications for constructing and maintaining the deployment manifest **212**, as well as other related documents and resources. In this implementation, the server **202** makes the application package **214** and the additional application resources **216** available over the network **205** to other computing devices.

The client computer **220** is a computing device configured to execute locally-running applications as well as connect to other computers over the network **205**. The client computer **220** also includes local storage in the form of a client data store **228**. On the client data store **228** resides an application store **230** and a transient cache **232**. In one embodiment, each application executing on client computer **220** has an associated application store **230**. The client computer **220** also includes other applications for interacting with other computers over the network. One such application is host software **222**, such as Internet browsing software (hereinafter referred to as browser **222**). The browser **222** communicates with an application package handler **224** and a resource loader **226**. The application package handler **224** is registered so that when the browser **222** encounters an application package, such as application package **214**, the browser knows to call the application package handler **224**. The application package handler **224** then processes the application package **214**. The application package **214** contains information to start execution of the associated

application on the client computer **220**. The format for the application package **214** may be an executable file or other packaging type format. In one embodiment, the application package handler **224** may be configured to decipher the format of the application package in order to obtain the relevant information. The browser **222** also communicates with the resource loader **226** when the client computer requests additional resources from the server **202**, such as additional application resources **216**. The processing performed by browser **222**, application package handler **224**, and resource loader **226** will be described in greater detail below in conjunction with flow diagrams FIGURES 5-7.

Briefly stated, a user of the client computer **220** may connect to the server **202** in any conventional manner. The server **202** presents a Web page or some other resource that makes available files that reside on the server data store **210**. In response to a selection of a link or the like by the user, the server **202** navigates to the deployment manifest **212**, which identifies the application package **214** associated with the requested application. As will be described in greater detail below, the application package **214** contains the minimum amount of code necessary to start the application. The application package **214** is brought down to the client computer **220** from the server **202**.

FIGURE 3 is an illustrative screen display that may be presented by Web browsing software enabling the progressive download of a remote application, in accordance with one implementation of the invention. Turning briefly to FIGURE 3, an example display **300** of the browser **222** is shown including a Web page **310** that may be served by the server **202** described above. The Web page **310** may be a resource associated with a particular Web app or may be a resource for making available software applications to remote computing systems for download. The Web page **310** includes a hyperlink **360** pointing to the deployment manifest **212** described above. The deployment manifest **212** points to the application package **214**, which contains at least the minimum code necessary to start the application. The Web page **310** also includes a hyperlink **380** pointing to the deployment manifest **212** described above. The selection of hyperlink **380** indicates that the user is now interested in explicitly "installing" the

application. As will be described in detail below in conjunction with FIGURE 7, upon selecting hyperlink **380**, the user may continue to interact with the application without waiting for the application to download or become installed on the computer.

It will be appreciated that the Web page **310** may be provided over the Internet, a corporate intranet, or any other network-accessible location. Activating the hyperlink **360** causes the application package **214** to be pulled down from the server. It should be appreciated that the Web page **310** is only one way that the user may invoke the application. For instance, a link to the application package **214** may be provided in an e-mail message, or the like.

Illustrative Techniques

FIGURE 4 is a state diagram **400** illustrating various states of the progressive installation of an application, in accordance with one implementation of the invention. The progressive installation includes an invoke state **402**, a start-up state **404**, a demand state **406**, and an installed state **410**. At the invoke state **402**, a user as invoked the application. If the user invokes the application by clicking a link that is served by server **202**, the progressive installation proceeds to the start-up state **404**. However, as will be described later in detail, the application may have become installed on the client computer already. The locally installed application may be invoked, such as by selecting a link to the local application, selecting a short-cut in the start menu to the local application, and the like. When the locally installed application is invoked, the process transitions to the installed state **410**. In another embodiment, the transition to the installed state **410** from the invoke state **402** may proceed via a subscription update state **412**. Briefly, the subscription update state **412** determines whether there is an update to the application available on the server **202**. If there are any updates, the updated components of the application are downloaded.

Now, assuming that the application has not been installed locally, the progressive installation proceeds to the start-up state **404**. Briefly, described in detail in conjunction with FIGURE 5, the start-up state **404** downloads the minimum code necessary for the application to run on the client computer. Because, the minimum code is considerably less than the full application, the user can begin interacting with the

application right away, similar to a user's experience when interacting with a traditional Web app today. From the start-up state, the progressive installation proceeds to the demand state **406**. Briefly, described in detail in conjunction with FIGURE 6, the demand state **406** downloads resources as needed. This allows the user to try the application before committing to purchasing the application or developing a lasting relationship with the application.

From the demand state **406**, the user may proceed to the exit state **408**, foregoing the installation of the application locally. This may occur when the user closes the browser. Once the user transitions from the demand state **406** to the exit state **408**, the downloaded components of the application may be deleted. Therefore, the client computer is in the same state as before the user invoked the application. The user may then later invoke the remote application again. The progressive installation will proceed again through the startup state and demand state. Thus, the user may use the application again without committing to installing the application. From the demand state **406**, the progressive installation may transition to the installed state **410**. The transition may be user-initiated based on a purchasing decision, a request for elevated permission (e.g., trust elevation), or may be performed autonomous by the operating system, such as when a pre-determined number of resources have already been installed on the client computer. The transition from the demand state to the installed state does not impact the user's interaction with the application. This transition is described below in conjunction with FIGURE 7.

Thus, the progressive installation in accordance with the present invention allows a user to begin interacting with an application as soon as it is invoked. Pieces of the application are downloaded as the user interacts without impacting the user. At no time, does the user need to wait for a dedicated installation.

FIGURE 5 is a logical flow diagram generally illustrating a process during a start-up state of the progressive installation in accordance with one embodiment of the present invention. The process begins at block **501** where an application residing over a network as been invoked, such as by selecting a hyperlink

related to the application. Before continuing with FIGURE 5, the components of the application residing over the network are described in conjunction with FIGURE 8.

FIGURE 8 is a graphical depiction of the components of the application residing over the network on server **202**. The components include the deployment manifest **212**, the application package **214**, and additional application resources **216**. The application package **214** includes an application manifest **802** and code **804**. The application manifest **802** describes the application code in detail including each component, their versions and dependencies. A sample application manifest of such a nature is included with this document as "Appendix A – Sample Application Manifest." Although described in this document as a specific file, the application manifest 212 of the invention should be interpreted to mean information describing the components of the application in any form and it may exist in locations other than just those described here. The application manifest **212** described here is by way of illustration only. In one embodiment, the code **804** includes the minimal code necessary to run the application. As one skilled in the art will appreciate, additional, non-necessary code may be included within application package **214** without departing from the present invention. However, in order to have the less delay or impact to the user, the minimal amount of code is desired. The additional application resources **216** include on demand resources and online resources, such as markup A **810**, additional code **812**, markup B **814** and the like. While FIGURE 8, illustrates only five additional resources, one skilled in the art will appreciate that, typically, there are several additional resources that are components of the Web app.

Returning to FIGURE 5, at block **502**, navigation to the deployment manifest **212** occurs in one embodiment of the present invention. In one embodiment, the deployment manifest resides on a remote server and identifies an entry point for the application. A sample deployment manifest is included with this document as "Appendix B – Sample Deployment Manifest."

At block **504**, navigation to the entry point occurs. In one embodiment, the entry point may be an application package (e.g., application package **214** shown in

FIGURE 8) that includes an application manifest and the minimum amount of code necessary to run the application.

At block **506**, the host is initiated. In one embodiment, the host is a web browser. In another embodiment, in which the application is being progressively installed from the client computer, the host may be a standalone host. The standalone host will then function in the same manner as described below using the embodiment in which the host is a web browser. The application package handler is registered for the file type associated with the application package. Therefore, when the browser receives the application package, the application package handler may initiate the progressive installation in accordance with the present invention. In one embodiment, the application package handler may be a mime handler that is registered for the file type associated with the application package **214**.

At block **508**, the minimum amount of code necessary to run the application is downloaded. For the embodiment using the application package **214**, this includes downloading the application package **214**. As mentioned above, the browser will call the application package handler to process the application package **214**.

At block **510**, the resource loader is registered and associated with the application. The resource loader is responsible for loading application resources as they are needed. In one embodiment, the resource loader may be a pluggable protocol that knows how to load resources "in the context" of an application.

At block **512**, an application domain for the application is created. At block **514**, the user code that constitutes the application is executed. In one embodiment, the execution of the application creates an application object. Code within the application package that defines a class is executed to create the application object. The application object has a unique identity. In addition, the application object includes a state. The state will be continuously updated during the demand state. The state includes information related to the user's interaction with the application. This state information will allow the application to smoothly transition from a web application to a client application.

Processing in the start-up state is complete. The progressive installation will then proceed to the demand state. As shown in FIGURE 8, after the start-up is complete, the application package **214** is stored in the application store **230** on the client computer **220**. The user may begin interacting with the application. For prior Web apps, resources of the web app were downloaded to the transient cache **232** without having the concept of a per application store **230**. As will be shown, by having the per application store **230**, the present invention may smoothly transition from a web app to a client-side application without impacting the user.

FIGURE 6 is a logical flow diagram generally illustrating a process during a demand state of the progressive installation. The demand state begins at block **601** where the startup state is completed and the user is interacting with the application. Process **600** depicts processing that occurs whenever a part of the application is requested. This may includes requests for assembly loads (code), resources, and the like. Typically, many requests for resources and code will be received during the demand state. Each such request will perform process **600**. The following discussion describes process **600** when the request is a resource. Those skilled in the art will appreciate that process **600** is also performed when the request is for an assembly load.

At block **602**, a request is received. Typically, the request occurs whenever a user selects a hyperlink on one of the web pages associated with the application. Processing continues to decision block **604**.

At decision block **604**, a determination is made whether the request is for a resource. If the request is not for a resource, processing proceeds to the end. On the other hand, when the request is for a resource, processing continues to decision block **606**.

At decision block **606**, a determination is made whether the requested resource is available locally. If the resource is available locally, the local copy of the resource is loaded for use and the process proceeds to the end. Depending on the type of resource, the local copy may either be in the application store or in the transient cache. If the resource is not available locally, processing continues at decision block **610**.

At decision block **610** a determination is made whether the requested resource is an on demand resource. If the requested resource is an on demand resource, processing continues to block **612** where the resource is loaded via http and cached in the local application store. For example, in FIGURE 9, markup A **810** and code **812** are stored in the application store **230**. The process proceeds to the end. At decision block **610**, if the resource is not an on demand resource, processing continues to decision block **620**.

At decision block **620**, a determination is made whether the resource is an online resource. If the resource is an online resource, processing continues to block **622** where the online resource is loaded via http. At block **624**, the online resource is cached in the transport cache **232**. For example, in FIGURE 9, markup b **814,** which is designated as an online resource, is stored in the transient cache. Processing is then complete.

In one embodiment, each resource may belong to a group of resources that are related in some fashion. For instance, resources that are commonly used together may be included in a group. In such a case, at blocks **612** and **622** the entire group of resources including the subject resource may be retrieved. This technique improves the likelihood that another resource that will be needed later will already exist locally.

Thus, as one will note, during the demand state, additional resources are downloaded and are populated in the per application store. Because these resources that are download are the same resources that are needed to run the application offline, as will be described below, the application store, along with the application object, allow the present invention to smoothly transition from a web application to a client-side application without impacting the user's interaction with the application. Thus, instead of having two different types of application, (i.e., a client-side application and a web application), one type of application may be used for both purposes. Using the present invention, the one type of application smoothly transitions from one purpose to the other purpose when desired.

FIGURE 7 is a logical flow diagram generally illustrating a process for transitioning between the demand state and the install state of the progressive installation. Processing begins at block **701** where a trigger has occurred to signal that the application should be installed. The trigger may be user-initiated or may be autonomous based on an external benchmark, such as the number of resources that have already been downloaded into the per application store. Processing continues at block **702**.

At block **702**, the remaining on demand resources are downloaded via http. Processing continues at block **704**. At block **704**, these remaining on demand resources are stored in the application store. This occurs while the user is still interacting with the application. For example, FIGURE 10 illustrates Markup C **816** now residing in the application store. Processing continues at block **706**.

At block **706**, a copy of online resources is stored in the application store **230**. Therefore, if the copy in the transient cache is removed, a copy of the online resource still exists. For example, referring to FIGURE 10, Markup B **814** is illustrated as being stored in the application store. Processing continues at block **708**.

At block **708**, activation information is recorded in the operating system. For example, a shortcut may be added to the start menu. The activation information allows the application to be invoked using traditional mechanisms the next time the application is invoked locally. Processing continues at block **710**.

At block **710**, impression information is recorded in the operating system. The impression information describes the manner in which the application interacts with the operating system, such as file associations. In addition, the impression information describes how the application may be changed/removed from the program entries. Referring to FIGURE 10, activation information **832** and impression information **834** are shown within operating system information **830** on the client computer **220**. Processing is then complete.

At mentioned above, at this point, the application is available offline. As one notes after reading the above description, the user did not have to wait for the installation of the application. Information generated during the demand state is

transitioned to the installed state. Thus, through the use of the application identity and state information that was stored while the user was interacting with the application, the application may smoothly transition to the install state.

Illustrative Application Programming Interface

In one specific example, the above-described techniques may be implemented with one or more Application Programming Interfaces (APIs) that expose functionality for managing the details of download and install, servicing, establishment of trust an privacy, and ultimate execution of the application. One example of such an API is described in the following example class for a DeploymentManager for handling each of these functions. What follows is a skeleton type-definition for this DeploymentManager:

```
* * * * * * * * * * * * * * * * * * * * * * * *
public class DeploymentManager
{
    public DeploymentManager(string identity, string codebase)
    { ... }

    // The events for the async operations which can be invoked.
    public event BindCompletedEventHandler BindCompleted;
    public event DeterminePlatformRequirementsCompletedEventHandler
DeterminePlatformRequirementsCompleted;
    public event DetermineAuthorizationCompletedEventHandler
DetermineAuthorizationCompleted;
    public event SynchronizeCompletedEventHandler
SynchronizeCompleted;
    public event ExecuteCompletedEventHandler ExecuteCompleted;

    // ProgressChanged event for all async operations.
    public event DeploymentProgressChangedEventHandler
DeploymentProgressChanged;

    // BindAsync
```

```
        public void BindAsync(object userToken)
        { ... }

        // DetermineRequirementsAsync
        public void DeterminePlatformRequirementsAsync(object userToken)
        { ... }

        // DetermineAuthorizationAsync
        public void DetermineTrustAsync(object userToken)
        { ... }

        // SynchronizeAsync
        public void SynchronizeAsync(object userToken)
        { ... }
                .
        // ExecuteAsync
        public void ExecuteAsync(object userToken)
        { ... }

        // AsyncCancel
        public void AsyncCancel()
        { ... }
    }
```

* *

Note that the DeploymentManager exposes methods for five principal operations: (1) Manifest Binding, (2) Determination of Platform Requirements, (3) Determination of Authorization, (4) Synchronization, and (5) Execution. Each of those functions shall be briefly described here.

Manifest Binding

Manifest Binding is the process by which necessary manifest metadata about a deployed application is initially obtained for the purposes of install, servicing and activation. Binding generally begins with either a codebase to a deployment

manifest, or a deployed application identity, or possibly both. Binding retrieves the minimum set of manifest information for making subsequent decisions about the deployed application. Manifest binds may or may not require network connectivity depending on the context of the bind, the state of application store and the application being bound to. If an application is already deployed on the machine, a bind may succeed in a completely offline manner, with no network I/O.

Determination of Platform Requirements

Once binding is complete it becomes possible to query the install state of the client machine to determine whether the necessary platforms are present to run the application. Platforms may be identified as any software upon which the application depends but cannot be installed as part of the deployed application install. Deployed applications reference these dependencies in their application manifest. For instance, support may be provided for minimum version of operating system, minimum version of runtime environment, and specific version of a GAC-resident assembly. Whether the version is treated as minimum may depend on whether the assembly is marked as a platform or a library.

If platform requirements are satisfied, the application install continues. If platform requirements are not satisfied then a failure is returned along with specific information on which platform dependencies were not satisfied. The application install process then cannot proceed but may be repeated after action is taken to install the necessary platforms on the machine.

Determination of Authorization

Decisions about what trust, privacy and licensing is to be accorded to the application can also be made once Manifest Binding is complete, since this information may also be resident in the deployment and application manifests. These decisions are grouped under the general category of Authorization. Authorization may require user prompting, e.g., if the application cannot run in the default sandbox. Successful Authorization results in the generation of the set of permission grants, privacy policy

guarantees, license keys, etc necessary for the application to run on the client machine. This information may be cached so that the decision does not have to be repeated later when the application is activated. If Authorization fails then application install cannot proceed.

Synchronization

Upon successful completion of Platform and Authorization determination, the actual task of installing the application payload can begin. This process is known as synchronization. A synchronization operation may succeed by simply ensuring that the required version of the deployed application is already present on the machine. Alternatively, synchronization may involve a full download of the remotely deployed application (required as well as on-demand components, language packs) or only the minimum required subset of components necessary to launch the application (e.g., required components only). When downloads are incurred synchronization can be further subdivided into two phases: (1) a pure transport phase where the application payload is replicated into a temporary storage location on disk and (2) a commit operation, where the deployed application is transacted to store and made available for binding. Synchronization can also be used to download the optional or on-demand components of a existing installed application.

Execution

Once an application has successfully completed Platform and Authorization decisions, and its payload has been successfully synchronized and committed to store, the application may then be executed (launched or run). Execution may take place in a separate, stand-alone process or may use the existing caller's process to run. In both cases the execution host provides a secure execution environment for the application, possibly utilizing previously cached decisions previously resulting from calls to Authorization to avoid reprompting.

Client-Side Implementations

To receive notifications from DeploymentManager method invocations, such as asynchronous completion results, the client provides an implementation for the associated completion event handlers (e.g., BindCompletedEventHandler, etc). These are passed an associated completion event argument (e.g., BindCompletedEventArgs). To receive asynchronous progress results for these operations, the client provides an implementation for DeploymentProgressChangedEventHandler. This will be passed an argument (DeploymentProgressChangedEventArgs). What follows is a skeleton example of such an event-handler implementation:

```
            * * * * * * * * * * * * * * * * * * * * * * * *
// Bind event handler delegate and args.
public delegate void BindCompletedEventHandler(object sender,
BindCompletedEventArgs e);
public class BindCompletedEventArgs : AsyncCompletedEventArgs
{
    public BindCompletedEventArgs(Exception error, bool cancelled,
object userToken) : base(error, cancelled, userToken)
    { ... }
}

// DeterminePlatformRequirements event handler delegate and args.
public delegate void
DeterminePlatformRequirementsCompletedEventHandler(object sender,
DeterminePlatformRequirementsCompletedEventArgs e);
public class DeterminePlatformRequirementsCompletedEventArgs :
AsyncCompletedEventArgs
{
    public DeterminePlatformRequirementsCompletedEventArgs(Exception
error, bool cancelled, object userToken) : base(error, cancelled,
userToken)
    { ... }
}
```

```
// DetermineAuthorization event handler delegate and args.
public delegate void
DetermineAuthorizationCompletedEventHandler(object sender,
DetermineAuthorizationCompletedEventArgs e);
public class DetermineAuthorizationCompletedEventArgs :
AsyncCompletedEventArgs
{
    public DetermineAuthorizationCompletedEventArgs(Exception error,
bool cancelled, object userToken) : base(error, cancelled, userToken)
    { ... }
}

// Synchronize event handler delegate and args.
 public delegate void SynchronizeCompletedEventHandler(object sender,
SynchronizeCompletedEventArgs e);
public class SynchronizeCompletedEventArgs : AsyncCompletedEventArgs
{
    public SynchronizeCompletedEventArgs(Exception error, bool
cancelled, object userToken) : base(error, cancelled, userToken)
    { ... }
}

// Execute event handler delegate and args.
public delegate void ExecuteCompletedEventHandler(object sender,
ExecuteCompletedEventArgs e);
public class ExecuteCompletedEventArgs : AsyncCompletedEventArgs
{
    public ExecuteCompletedEventArgs(Exception error, bool cancelled,
object userToken) : base(error, cancelled, userToken)
    { ... }
}

// DeploymentProgressChanged event handler delegate and args.
public delegate void DeploymentProgressChangedEventHandler(object
sender, DeploymentProgressChangedEventArgs e);
public class DeploymentProgressChangedEventArgs :
ProgressChangedEventArgs
```

```
{
    public DeploymentProgressChangedEventArgs(object userToken, int
progressPercentage) : base(userToken, progressPercentage)
    { ... }
}
```

* *

What follows is a general example of one implementation of a client (e.g., an application) that invokes the mechanisms just described. The following example is an application based on a class (Client) that implements the DeploymentManager described above.

* *

```
public class Client
{
    public static void Main()
    {
        DeploymentManager dep = new DeploymentManager("name=bar,
version=1.0.0.0", "http://www.foo.com/bar.deploy");
        dep.DeploymentProgressChanged += new
DeploymentProgressChangedEventHandler(Client.ProgressChanged);
        dep.BindCompleted += new
BindCompletedEventHandler(Client.BindCompleted);

        dep.BindAsync(null);
    }
    public static void BindCompleted(object sender,
BindCompletedEventArgs e)
    {
        System.Console.WriteLine(e.ToString());
    }
    public static void ProgressChanged(object sender,
DeploymentProgressChangedEventArgs e)
    {
        System.Console.WriteLine(e.ProgressPercentage);
```

```
    }
}
```

* *

The above specification, examples and data provide a complete description of the structure and use of implementations of the invention. Since many embodiments of the invention can be made without departing from the spirit and scope of the invention, the invention resides in the claims hereinafter appended.

Appendix A. Sample Application Manifest

```
<?xml version="1.0" encoding="utf-8"?>

<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0" xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1
assembly.adaptive.xsd">

  <assemblyIdentity name="MyDocViewer" version="1.0.0.0"
processorArchitecture="x86" asmv2:culture="en-us"
publicKeyToken="0000000000000000" />

  <entryPoint name="main" xmlns="urn:schemas-microsoft-com:asm.v2"
dependencyName="MyDocViewer">
    <commandLine file="MyDocViewer.exe" parameters="" />
  </entryPoint>

  <TrustInfo xmlns="urn:schemas-microsoft-com:asm.v2"
xmlns:temp="temporary">
    <Security>
      <ApplicationRequestMinimum>
        <PermissionSet class="System.Security.PermissionSet"
ID="FullTrust" version="1" Unrestricted="true" />
        <AssemblyRequest name="MyDocViewer"
PermissionSetReference="FullTrust" />
      </ApplicationRequestMinimum>
    </Security>
  </TrustInfo>

  <file name="Sample4.xaml"
hash="75966580bf63a6f7d9818bcbf6c8c61343e61d9f" hashalg="SHA1"
asmv2:size="636" />
  <file name="Sample5.xaml"
hash="9fe4e7312498c0b62ab455b289e27fc2fc8b3bb3" hashalg="SHA1"
asmv2:size="615" />
  <file name="Sample6.xaml"
hash="760221281e78c621f45947b97b87e65a2bee6e14" hashalg="SHA1"
asmv2:size="2750" />

  <dependency asmv2:name="MyDocViewer">
    <dependentAssembly>
      <assemblyIdentity name="MyDocViewer" version="0.0.0.0"
publicKeyToken="f745653e2b97409d" processorArchitecture="x86" />
    </dependentAssembly>
    <asmv2:installFrom codebase="MyDocViewer.exe"
hash="95f2246ac74b3f32938db0ebed313e38ee7b4b5b" hashalg="SHA1"
size="12288" />
  </dependency>
</assembly>
```

Appendix B. Sample Deployment Manifest

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0" xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1
assembly.adaptive.xsd">
  <assemblyIdentity name="MyDocViewer.deploy" version="1.0.0.0"
processorArchitecture="x86" asmv2:culture="en-us"
publicKeyToken="0000000000000000" />
  <description asmv2:publisher="MyCompany" asmv2:product="WCP
Application of MyDocViewer"
asmv2:supportUrl="http://www.mycompany.com/AppServer/MyDocViewer/suppo
rt.asp" />
  <deployment xmlns="urn:schemas-microsoft-com:asm.v2"
isRequiredUpdate="false">
    <install shellVisible="true" />
    <subscription>
      <update>
        <beforeApplicationStartup />
        <periodic>
          <minElapsedTimeAllowed time="6" unit="hours" />
          <maxElapsedTimeAllowed time="1" unit="weeks" />
        </periodic>
      </update>
    </subscription>
  </deployment>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity name="MyDocViewer" version="1.0.0.0"
processorArchitecture="x86" asmv2:culture="en-us"
publicKeyToken="0000000000000000" />
    </dependentAssembly>
    <asmv2:installFrom codebase="MyDocViewer.manifest" />
  </dependency>
</assembly>
```